Micmatch Version 1.0.0
Reference Manual

Martin Jambon

July 21, 2008

This manual is available online as a single HTML file at
`http://martin.jambon.free.fr/micmatch-manual.html`
and as a PDF document at
`http://martin.jambon.free.fr/micmatch-manual.pdf`.
The home page of Micmatch is:
`http://martin.jambon.free.fr/micmatch.html`

Contents

1 Introduction

Micmatch is an extension of the syntax of the Objective Caml programming language (OCaml). Its purpose it to make the use of regular expressions easier and more generally to provide a set of tools for using OCaml as a powerful scripting language. Micmatch believes that regular expressions are just like any other program and deserve better than a cryptic sequence of symbols placed in a string of a master program.

Micmatch currently supports two different libraries that implement regular expressions: Str which comes with the original distribution of OCaml and PCRE-OCaml which is an interface to PCRE (Perl Compatible Regular Expressions) for OCaml. These two flavors will be referred as Micmatch_str and Micmatch_pcre. They share a large number of syntaxic features, but Micmatch_pcre provides several macros that cannot be implemented safely in Micmatch_str. Therefore, it is recommended to use Micmatch_pcre.

2 Language

2.1 Regular expressions

2.1.1 Grammar of the regular expressions

Regular expressions support the syntax of Ocamllex regular expressions as of version 3.08.1 of the Objective Caml system (`http://caml.inria.fr/ocaml/htmlman/`), and several additional features. A regular expression (*regexp*) is defined by the grammar that follows. The associativity rules are given by priority levels. 0 is the strongest priority.

- <u>*char-literal*</u> Match the given character (priority 0).
- `_` (underscore) Match any character (priority 0).

- *string-literal* Match the given sequence of characters (priority 0).
- [*set-of-characters*] Match one of the characters given by *set-of-characters* (priority 0). The grammar for *set-of-characters* is the following:
  - *char-literal*−*char-literal* defines a range of characters according to the iso-8859-1 encoding (includes ASCII).
  - *char-literal* defines a singleton (a set containing just this character).
  - *string-literal* defines a set that contains all the characters present in the given string.
  - *lowercase-identifier* is replaced by the corresponding predefined regular expression; this regular expression must be exactly of length 1 and therefore represents a set of characters.
  - *set-of-characters set-of-characters* defines the union of two sets of characters.
- *regexp* # *regexp* Match any of the characters given by the first regular expression except those which are given by the second one. Both regular expressions must be of length 1 and thus stand for a set of characters (priority 0).
- [^*set-of-characters*] Same as _ # [*set-of-characters*] (priority 0).
- *regexp* * Match the pattern given by *regexp* 0 time or more (priority 0).
- *regexp* + Match the pattern given by *regexp* 1 time or more (priority 0).
- *regexp*{*m*−*n*} Match *regexp* at least m times and up to n times. m and n must be integer literals (priority 0).
- *regexp*{*n*} Same as *regexp*{*n*−*n*} (priority 0).
- *regexp*{*n*+} Same as *regexp*{*n*}*regexp*∗ (priority 0).
- *regexp*{*n*−} Deprecated. Same as *regexp*{*n*+} (priority 0).
- (*regexp*) Match *regexp* (priority 0).
- *regexp* ~ Case insensitive match of the given regular expression *regexp* according to the conventions of Objective Caml, i.e. according to the representation of characters in the iso-8859-1 standard (latin1) (priority 0).
- *regexp regexp* Match the first regular expressions and then the second one (priority 1).
- *regexp* | *regexp* Match one of these two regular expressions (priority 2).
- *regexp* **as** *lowercase-identifier* Give a name to the substring that will be matched by the given pattern. This string becomes available under this name (priority 3). In-place conversions of the matched substring can be performed using one these three mechanisms:

– *regexp* **as** *lowercase-identifier* **:** *built-in-converter* where *built-in-converter* is one of `int`, `float` or `option`. `int` behaves as `int_of_string`, `float` behaves as `float_of_string`, and `option` encapsulate the substring in an object of type `string option` using an equivalent of `function "" -> None | s -> Some s`

– *regexp* **as** *lowercase-identifier* **:=** *converter* where *converter* is any function which converts a string into something else.

– *regexp* **as** *lowercase-identifier* = *expr* where *expr* is any OCaml expression, usually a constant, which assigns a value to *lowercase-identifier* without knowing which substring it matches.

- % *lowercase-identifier* Give a name to the position in the string that is being matched. This position becomes available as an int under this name.

- @ *expr* Match the string given by *expr*. *expr* can be any OCaml expression of type string. Parentheses will be needed around *expr* if it is a function application, or any construct of equivalent or lower precedence (see the Objective Caml manual, chapter "The Objective Caml language", section "Expressions").

2.1.2 Named regular expressions

Naming regular expressions is possible using the following toplevel construct:
RE *ident* = *regexp*
where *ident* is a lowercase identifier. Regular expressions share their own namespace.

For instance, we can define a phone number as a sequence of 3 digits followed by a dash and followed by 4 digits:

```
RE digit = ['0'-'9']
RE phone = digit{3} '-' digit{4}
```

2.1.3 Predefined sets of characters

The POSIX character classes (sets of characters) are available as predefined regular expressions of length 1. Their definition is given in table 1.

2.1.4 More predefined patterns

Some named regexps are predefined and available in every implementation of Micmatch. These are the following:

- `int`: matches an integer (see table 2). It accepts a superset of the integer literals that are produced with the OCaml standard function `string_of_int`.

- `float`: matches a floating-point number (see table 2). It accepts a superset of the float literals that are produced with the OCaml standard function `string_of_float`.

Table 1: POSIX character classes and their definition in the Micmatch syntax

```
RE lower = ['a'-'z']
RE upper = ['A'-'Z']
RE alpha = lower | upper
RE digit = ['0'-'9']
RE alnum = alpha | digit
RE punct = ["!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"]
RE graph = alnum | punct
RE print = graph | ' '
RE blank = ' ' | '\t'
RE cntrl = ['\x00'-'\x1F' '\x7F']
RE xdigit = [digit 'a'-'f' 'A'-'F']
RE space = [blank "\n\x0B\x0C\r"]
```

Table 2: Predefined regexps in Micmatch

```
RE int = ["-+"]? ( "0" ( ["xX"] xdigit+
                       | ["oO"] ['0'-'7']+
                       | ["bB"] ["01"]+ )
                 | digit+ )

RE float =
  ["-+"]?
     ( ( digit+ ("." digit* )? | "." digit+ ) (["eE"] ["+-"]? digit+ )?
       | "nan"~
       | "inf"~ )
```

2.2 General pattern matching

2.2.1 Regexps and match/function/try constructs

In Micmatch, regular expressions can be used to match strings instead of the regular patterns. In this case, the regular expression must be preceded by the **RE** keyword, or placed between slashes (/.../). Both notations are equivalent.

Only the following constructs support patterns that contain regular expressions:

- **match** ... **with** *pattern* **->** ...
- **function** *pattern* **->** ...
- **try** ... **with** *pattern* **->** ...

Examples:

```
let is_num = function RE ['0'-'9']+ -> true | _ -> false

let get_option () =
  match Sys.argv with
      [| _ |] -> None
    | [| _; RE (['a'-'z']+ as key) "=" (_* as data) |] -> Some (key, data)
    | _ -> failwith "Usage: myprog [key=value]"

let option =
  try get_option ()
  with Failure RE "usage"~ -> None
```

If alternatives are used in a pattern, then both alternatives must define the same set of identifiers. In the following example, the string `code` can either come from the normal pattern matching or be a fresh substring which was extracted using the regular expression:

```
match option, s with
    Some code, _
  | None, RE _* "=" (['A'-'Z']['0'-'9'] as code) -> print_endline code

  | _ -> ()
```

In the general case, it is not possible to check in advance if the pattern-matching cases are complete if at least one of the patterns is a regular expression. In this case, no warnings against missing cases are displayed, thus it is safer to either add a catch-all case like in the previous examples or to catch the `Match_failure` exception that can be raised unexpectedly.

2.2.2 Views (experimental feature)

Views are a general form of symbolic patterns other than those authorized by the concrete structure of data. For example, `Positive` could be a view for positive ints. View patterns can also bind variables and a useful example in OCaml is pattern-matching over lazy values.

Here we propose simple views, as suggested by Simon Peyton Jones for Haskell: `http://hackage.haskell.org/trac/ghc/wiki/ViewPatterns`. We propose a different syntax, but note that the syntax that we have chosen here is experimental and may change slightly in future releases.

2.2.2.1 View patterns

A view pattern has one of these two forms:

1. % *view-name*: a view without an argument. It is a simple check over the subject data.
2. % *view-name pattern*: a view with an argument, the pattern. If the view function matches successfully, its result is matched against the given pattern.

where a *view-name* is a capitalized alphanumeric identifier, possibly preceded by a module path specification, e.g. `Name` or `Module.Name`.

2.2.2.2 Definition of a view

Views without arguments are defined as functions of type `'a -> bool`, while views with arguments are defined as functions of type `'a -> 'b option`.

The syntax for defining a view is:

- **let view** *uppercase-identifier* = *expression*
- **let view** *uppercase-identifier* = *expression* **in** *expression*

Using the syntax above is however not strictly needed, since it just defines a function named after the name of the view, and prefixed by `view_`. For instance `let view X = f` can be written as `let view_X = f` in regular OCaml. Therefore, some library modules can export view definitions without using any syntax extension themselves.

2.2.2.3 Example

```
(* The type of lazy lists *)
type 'a lazy_list = Nil | Cons of ('a * 'a lazy_list lazy_t)

(* Definition of a view without argument for the empty list *)
let view Nil =
    fun l ->
      try Lazy.force l = Nil
      with _ -> false

(* Independent definition of a view with an argument,
   the head and tail of the list *)
let view Cons =
    fun l ->
```

```
      try
        match Lazy.force l with
            Cons x -> Some x
          | Nil -> None
      with _ -> None


(* Test *)
let _ =
  let l = lazy (Cons (1, lazy (Cons (2, lazy Nil)))) in
  match l with
      %Nil
    | %Cons (_, %Nil) -> assert false
    | %Cons (x1, %Cons (x2, %Nil)) ->
        assert (x1 = 1);
        assert (x2 = 2);
        Printf.printf "Passed view test\n%!"
    | _ -> assert false
```

2.2.2.4 Limitations

Each time a value is tested against a view pattern, the corresponding function is called. There is no optimization that would avoid calling the view function twice on the same argument.

Redundant or missing cases cannot be checked, just like when there is a regexp in a pattern. This is due both to our definition of views and to the implementation that we get using Camlp5.

2.3 Shortcut for one-case regexp matching

A shortcut notation can be used to extract substrings from a string that match a pattern which is known in advance:
let /*regexp*/ = *expr* **in** *expr*

Global declarations also support this shortcut:
let /*regexp*/ = *expr*
Example:

```
# Sys.ocaml_version;;
- : string = "3.08.3"
# RE int = digit+;;
# let /(int as major : int) "." (int as minor : int)
       ("." (int as patchlevel) | ("" as patchlevel))
       ("+" (_* as additional_info) | ("" as additional_info))/ =
    Sys.ocaml_version
;;
val additional_info : string = ""
val major : int = 3
```

```
val minor : int = 8
val patchlevel : string = "3"
```

The notation does not allow simultaneous definitions using the **and** keyword nor recursive definitions using **rec**.

As usual, the `Match_failure` exception is raised if the string fails to match the pattern. The let-try-in-with construct described in the next section also supports regexp patterns, with the same restrictions.

2.4 The let-try-in-with construct

A general notation for catching exceptions that are raised during the definition of bindings is provided:

let try [**rec**] *let-binding* {**and** *let-binding*} **in**
  expr
with *pattern-matching*

It has the same meaning as:

try let [**rec**] *let-binding* {**and** *let-binding*} **in**
  expr
with *pattern-matching*

except that in the former case only the exceptions raised by the *let-binding*s are handled by the exception handler introduced by **with**.

2.5 Implementation-dependent features

These features depend on which library is actually used internally for manipulating regular expressions. Currently two libraries are supported: the Str library from the official OCaml distribution and the PCRE-OCaml library. Support for other libraries might be added in the future.

2.5.1 Backreferences

Previously matched substrings can be matched again using backreferences. **!***ident* is a backreference to the named group *ident* that is defined previously in the sequence. During the matching process, it is not possible that a backreference refers to a named group which is not matched. In the following example, we extract the repeated pattern `abc` from `abcabc`:

```
# match "abcabc" with RE _* as x !x -> x;;
- : string = "abc"
```

2.5.2 Specificities of Micmatch_str

Backreferences as described previously (section 2.5.1) are supported.

In addition to the POSIX character classes, a set of predefined patterns is available:

- bol matches at beginning of line (either at the beginning of the matched string, or just after a newline character).

- eol matches at end of line (either at the end of the matched string, or just before a newline character).
- any matches any character except newline.
- bnd matches word boundaries.

2.5.3 Specificities of Micmatch_pcre

This is currently the version which is used by the `micmatch` command.

2.5.3.1 Matching order

Alternatives (*regexp1*|*regexp2*) are tried from left to right.

The quantifiers (`*`, `+`, `?` and `{...}`) are greedy except if specified otherwise (see next paragraph). The regular expressions are matched from left to right, and the repeated patterns are matched as many times as possible before trying to match the rest of the regular expression and either succeed or give up one repetition before retrying (backtracking).

2.5.3.2 Greediness and laziness Normally, quantifiers (`*`, `+`, `?` and `{...}`) are greedy, i.e. they perform the longest match in terms of number of repetitions before matching the rest of the regular expression or backtracking. The opposite behavior is laziness: in that case, the number of repetitions is made minimal before trying to match the rest of the regular expression and either succeed or continue with one more repetition.

The lazy behavior is turned on by placing the keyword `Lazy` after the quantifier. This is the equivalent of Perl's quantifiers `*?`, `+?`, `??` and `{...}?`. For instance, compare the following behaviors:

```
# match "<hello><world>" with RE "<" (_* as contents) ">" -> contents;;
- : string = "hello><world"
# match "<hello><world>" with RE "<" (_* Lazy as contents) ">" -> contents;;
- : string = "hello"
```

2.5.3.3 Possessiveness or atomic grouping Sometimes it can be useful to prevent backtracking. This is achieved by placing the `Possessive` keyword after a given group. For instance, compare the following:

```
# match "abc" with RE _* _ -> true | _ -> false;;
- : bool = true
# match "abc" with RE _* Possessive _ -> true | _ -> false;;
- : bool = false
```

This operator has the strongest associativity priority (0), just like the quantifiers.

2.5.3.4 Backreferences Backreferences are supported as described in section 2.5.1.

2.5.3.5 Predefined patterns The following predefined patterns are available in addition to the POSIX character classes:

- bos matches at beginning of the matched string.
- eos matches at the end of the matched string.
- bol matches at beginning of line (either at the beginning of the matched string, or just after a newline character).
- eol matches at end of line (either at the end of the matched string, or just before a newline character).
- any matches any character except newline.

2.5.3.6 Lookaround assertions

A lookaround assertion is a pattern that has to be matched but doesn't consume characters in the string being matched.

Lookahead assertions are checked after the current position in the string, and lookbehind assertions are matched before the current point. The general syntax for an assertion is the following:

< *lookbehind* . *lookahead* >

< *lookahead* >

The central dot symbolizes the current position. The *lookbehind* assertion is a test over the characters at the left of the current point, while the *lookahead* is a test over the characters at the right of the current point in the string.

lookbehind or *lookahead* are either empty or a regular expression, optionally preceded by `Not`. An assertion starting with `Not` is called negative and means that the given regular expression can not match here.

There are no restrictions on the contents of lookahead regular expressions. Lookbehind regular expressions are restricted to those that match substrings of length that can be predetermined. Besides this, backreferences are not supported in lookbehind expressions.

2.5.3.7 Macros This implementation provides a set of macros that follow this syntax:

MACRO-NAME regexp `->` *expr*

where *expr* is the expression that will be computed every time the pattern given by *regexp* is matched.

Only the `SPLIT` and `FILTER` macros follows a simplified syntax:

MACRO-NAME regexp

These constructs build a function which accepts some optional arguments and the string to match. For instance,

`(REPLACE "," -> ";") "a,b,c"`

returns `"a;b;c"` whereas

`(REPLACE "," -> ";") ~pos:2 "a,b,c"`

returns `"a,b;c"`

The possible options are the following:

- `pos` has type `int` and indicates that matching or searching must start from this position in the string. Its default value is always 0 (beginning of the string).

- `full` is a boolean that defines whether split operations must ignore empty fragments before the first matched pattern or the last matched pattern in the string. The default value is `true` for `MAP` and `false` for `SPLIT`.

- `share` is a potentially unsafe option which allows the reuse of some mutable data which are associated to a given regular expression. This may make the program slightly faster, but should generally not be used in multi-threaded programs or in libraries.

`MATCH` *regexp* `->` *expr*
tries to match the pattern *regexp* at the beginning of the string or at the given position `pos` and returns *expr* or raise `Not_found`. Options: `pos` (0), `share` (false). When `pos` and `share` are not specified, it is equivalent to:

```
function
    RE regexp -> expr
  | _ -> raise Not_found
```

`REPLACE` *regexp* `->` *expr*
returns a string in which every occurrence of the pattern is replaced by *expr*. Options: `pos` (0).

`REPLACE_FIRST` *regexp* `->` *expr*
returns a string in which the first occurrence of the pattern is replaced by *expr*. A copy of the input string is returned if the pattern is not found. Options: `pos` (0).

`SEARCH` *regexp* `->` *expr*
simply evaluates *expr* every time the pattern is matched. Options: `pos` (0).

`SEARCH_FIRST` *regexp* `->` *expr*
simply evaluates *expr* the first time the pattern is matched and returns the result. Exception `Not_Found` is raised if the pattern is not matched. Options: `pos` (0), `share` (false).

`COLLECT` *regexp* `->` *expr*
evaluates *expr* every time the pattern is matched and puts the result into a list. Options: `pos` (0).

`COLLECTOBJ` *regexp*
like `COLLECT`, but the elements of the returned list are automatically objects with methods that correspond to the subgroups captured with `as`. Options: `pos` (0).

`SPLIT` *regexp*
splits the given string using *regexp* as a delimiter. Options: `pos` (0), `full` (false).

`FILTER` *regexp*
creates a predicate that returns true is the given string matches *regexp* or false otherwise. Options: `pos` (0), `share` (false).

`CAPTURE` *regexp*
returns `Some o` where `o` is an object with methods that correspond to the captured subgroups, or `None` if the subject string doesn't match *regexp*. Options: `pos` (0), `share` (false).

`MAP` *regexp* `->` *expr*

splits the given string into fragments: the fragments that do not match the pattern are returned as `` `Text s `` where s is a string. Fragments that match the pattern are replaced by the result of *expr*, which has to be a polymorphic variant. Options: `pos` (0), `full` (true). For instance,

```
(MAP ',' -> `Sep) "a,b,c,"
```

returns the list

```
[`Text "a"; `Sep; `Text "b"; `Sep; `Text "c"; `Sep; `Text ""]
```

whereas

```
(MAP ',' -> `Sep) ~full:false "a,b,c,"
```

returns only

```
[`Text "a"; `Sep; `Text "b"; `Sep; `Text "c"; `Sep]
```

3 Tools

3.1 The toplevel

3.1.1 Micmatch_str

The `micmatch_str` command can be used as a replacement for `ocaml` either as an interactive toplevel or for executing scripts. Any library which is required by Micmatch_str is automatically loaded.

3.1.2 Micmatch_pcre

The `micmatch` command can be used as a replacement for `ocaml` either as an interactive toplevel or for executing scripts. Any library which is required by Micmatch_pcre is automatically loaded.

3.2 The libraries for the preprocessor

3.2.1 Micmatch_str

The preprocessing library `pa_micmatch_str.cma` must be loaded by the preprocessor (`camlp5o` or `camlp5r`).

It is safe to use Micmatch_str in multithreaded programs only if the Str library is not being used by one thread when some other threads are using Micmatch. When compiling multithreaded programs, the `-thread` option must be passed to the preprocessor.

3.2.2 Micmatch_pcre

The preprocessing library `pa_micmatch_pcre.cma` must be loaded by the preprocessor (`camlp5o` or `camlp5r`). When compiling multithreaded programs, the `-thread` option must be passed to the preprocessor (versions earlier than 0.693 do not require the `-thread` flag and do not accept it anyway. It is strictly required only for regexps with gaps (@expr) which did not exist before).

3.3 The runtime libraries

Both variants depend on portable features of the Unix library. The executables must therefore be linked against `unix.cma` (bytecode) or `unix.cmxa` (native code) in addition to the specific libraries mentioned below.

3.3.1 Micmatch_str

In addition to the backend for the regular expressions engine (`str.cma` for bytecode or `str.cmxa` for native code), the OCaml code which is produced by the preprocessor needs to be linked against either `run_micmatch_str.cma` (bytecode), `run_micmatch_str.cmxa` (native code), `run_micmatch_str_mt.cma` (bytecode, threads) or `run_micmatch_str_mt.cmxa` (native code, threads).

3.3.2 Micmatch_pcre

In addition to the backend for the regular expressions engine (`pcre.cma` for bytecode or `pcre.cmxa` for native code), the OCaml code which is produced by the preprocessor needs to be linked against either `run_micmatch_pcre.cma` (bytecode), `run_micmatch_pcre.cmxa` (native code). Multithreaded programs are supported as well and do not require a specific library.

4 Module `Micmatch` : A small text-oriented library

The `Micmatch` module provides a submodule named `Text`. A normal usage is to place `open Micmatch` at the beginning of user code that uses it.

This module is part of the runtime environment of Micmatch (the library run_micmatch_pcre.cma or equivalent).

```
module Text :
  sig
```

This module provides some general functions which are especially useful for manipulating text and text files.

```
val iter_lines_of_channel : (string -> unit) -> Pervasives.in_channel -> unit
```

`iter_lines_of_channel f ic` reads input channel `ic` and applies successively the given function `f` to each line until the end of file is reached.

```
val iter_lines_of_file : (string -> unit) -> string -> unit
```

`iter_lines_of_file f file` reads file `file` and applies successively the given function `f` to each line until the end of file is reached.

```
val lines_of_channel : Pervasives.in_channel -> string list
```

`lines_of_channel ic` returns the list of the lines that can be read from input channel `ic`.

`val lines_of_file : string -> string list`

`lines_of_file file` returns the list of the lines that can be read from file `file`.

`val channel_contents : Pervasives.in_channel -> string`

`channel_contents ic` returns the string containing the bytes that can be read from the given input channel `ic`.

`val file_contents : ?bin:bool -> string -> string`

`file_contents file` returns the string containing the bytes that can be read from the given file. Option `bin` specifies if `Pervasives.open_in_bin` should be used instead of `Pervasives.open_in` to open the file. Default is `false`.

`val save : string -> string -> unit`

`save file data` stores the string `data` in `file`. If the file already exists, its contents is discarded silently.

`val save_lines : string -> string list -> unit`

`save_lines file l` saves the given list `l` of strings in `file` and adds a newline characters (`'\n'`) after each of them. If the file already exists, its contents is discarded silently.

`exception Skip`

This exception can be used to skip an element of a list being processed with `rev_map`, `map`, `fold_left`, and `fold_right`.

`val map : ('a -> 'b) -> 'a list -> 'b list`

Like `List.map` but it is guaranteed that the elements of the input list are processed from left to right. Moreover the `Skip` exception can be used to skip an element of the list. This function runs in constant stack space.

`val rev_map : ('a -> 'b) -> 'a list -> 'b list`

Like `List.rev_map`, but it is guaranteed that the elements of the input list are processed from left to right. Moreover the `Skip` exception can be used to skip an element of the list. This function runs in constant stack space and is slightly faster then `map`.

`val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

Like `List.fold_left` but the `Skip` exception can be used to skip an element of the list. This function runs in constant stack space.

`val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

Like `List.fold_right` but the `Skip` exception can be used to skip an element of the list. This function runs in constant stack space.

```
val map_lines_of_channel : (string -> 'a) -> Pervasives.in_channel -> 'a list
```

`map_lines_of_channel f ic` is equivalent to `map f (lines_of_channel ic)` but faster.

```
val map_lines_of_file : (string -> 'a) -> string -> 'a list
```

`map_lines_of_file f file` is equivalent to `map f (lines_of_file file)` but faster.

```
end
```

```
module Fixed :
  sig
```

This module provides some functions which are useful for manipulating files with fields of fixed width.

```
val chop_spaces : string -> string
```

`chop_spaces s` returns a string where the leading and trailing spaces are removed.

```
val int : string -> int
```

`int s` reads an int from a string where leading and trailing spaces are allowed. Equivalent to `Pervasives.int_of_string (chop_spaces s)`.

```
val float : string -> float
```

`float s` reads an float from a string where leading and trailing spaces are allowed. Equivalent to `Pervasives.float_of_string (chop_spaces s)`.

```
end
```

```
module Directory :
  sig
```

Basic operations on directories

```
val list : ?absolute:bool -> ?path:bool -> string -> string list
```

`list dir` returns the alphabetically sorted list of the names of the files contained in directory `dir`. The special names that refer to the parent directory (e.g. `..`) and the directory itself (e.g. `.`) are ignored.

If the option `absolute` is set to `true`, the result is a list of absolute file paths, i.e. that do not depend on the current directory which is associated to the process (default is false; implies `path = true`).

If the option `path` is set to `true`, the result is a list of paths instead of just the file names (default is `false` except if `absolute` is explicitely set to `true`).

Exception `Invalid_argument "Directory.list"` is raised if there is an incompatibility between the options. Unspecified exceptions will be raised if the given directory does not exist or is not readable.

```
val is_dir : ?nofollow:bool -> string -> bool
```

`is_dir dir` returns true if `dir` is a directory, false otherwise. The `nofollow` option is false by default, but if true, a symbolic link will not be followed. In that case false is returned even if the link points to a valid directory.

```
  end
module Glob :
  sig
```

A generic file path matching utility

```
val scan :
  ?absolute:bool ->
  ?path:bool ->
  ?root:string ->
  ?nofollow:bool -> (string -> unit) -> (string -> bool) list -> unit
```

`scan action path_filter` returns all the file paths having a name that matches `path_filter`. `path_filter` is a list of filters that test whether a directory name or a file name should be selected.

The path search starts from the current directory by default, or from the directory specified by the `root` option. The file names are examined in an undefined order. When a file path matches, `action` is applied to the string representing the path. Options `absolute` and `path` have the same meaning and the same default values as in `Micmatch.Directory.list`[4].

`nofollow` can be used to prevent from considering symbolic links as directories. It is false by default. See also `Micmatch.Directory.is_dir`[4].

```
val lscan :
  ?rev:bool ->
  ?absolute:bool ->
  ?path:bool ->
  ?root:string list ->
  ?nofollow:bool -> (string list -> unit) -> (string -> bool) list -> unit
```

Same as `Micmatch.Glob.scan`[4] but file paths are kept as a list of strings that form a valid path when concatenated using `Filename.concat`. Option `rev` can be set if the lists representing paths are in reversed order, i.e. the root comes last.

In `lscan action path_filter`, options `rev`, `absolute`, and `path` take their default values which are all false. In this situation, it is guaranteed that the paths that are passed to `action` have the same length as `path_filter`.

```
val list :
  ?absolute:bool ->
  ?path:bool ->
  ?root:string ->
  ?nofollow:bool -> ?sort:bool -> (string -> bool) list -> string list
```

`list path_filter` works like `Micmatch.Glob.scan`[4] but returns a list of all file paths that match `path_filter`.

An example in Micmatch syntax is `list [FILTER _* ".ml" eos]`. It returns the list of ".ml" files in the current directory. It could have been written as `list [ fun s -> Filename.check_suffix s ".ml"]` and is equivalent to `*.ml` in shell syntax.

```
val llist :
  ?rev:bool ->
  ?absolute:bool ->
  ?path:bool ->
  ?root:string list ->
  ?nofollow:bool -> ?sort:bool -> (string -> bool) list -> string list list
```

`llist path_filter` works like `Micmatch.Glob.lscan`[4] but returns a list of all file paths that match `path_filter`.

```
end
```